TP 2: MyInt

NB: pour toutes vos fonctions vous devez bien préciser leurs types !

1 Une nouvelle version de Integer

Haskell nous offre deux types qui représentent les entiers : `Int` et `Integer`. La différence entre ces deux types est que les données de type `Int` ont une valeur maximum bornée (2^{63}), alors que les données de type `Integer` peuvent stocker (théoriquement) n'importe quel entier.

Pourquoi est-ce que Haskell nous offre deux types qui font la même chose ? En termes pratiques, les deux types ont une différence d'efficacité : le type `Int` correspond au type `int` de C, et représente des entiers sur lesquels l'ordinateur peut effectuer des opérations arithmétiques avec une seule instruction. Cela implique la borne supérieure (le processeur peut manipuler 64 bits maximum pour chaque opération). À l'autre côté, le type `Integer` utilise des algorithmes et une représentation des entiers basée sur des tableaux. Par conséquent, ce type peut représenter des entiers sans borne supérieur sur la valeur, mais les opérations arithmétiques sont moins efficaces.

Par exemple, on peut considérer les deux fonctions suivantes :

```
fact1:: Int -> Int
fact1 0  = 1
fact1 n  = n*(fact1 (n-1))

fact2:: Integer-> Integer
fact2 0  = 1
fact2 n  = n*(fact2 (n-1))
```

Si on les teste on a

```
*Main> fact1 50
-3258495067890909184
*Main> fact2 50
30414093201713378043612608166064768844377641568960512000000000000
```

On observe donc le problème d'**overflow** associé avec le type `Int`.

L'objectif de cet exercice est de programmer un type `MyInt` qui réimplémente la fonctionnalité du type `Integer`, c'est-à-dire, un type qui peut représenter des entiers de n'importe quelle valeur et effectuer des opérations arithmétiques sur ces entiers.

Comment faire ?

Vous avez le droit d'utiliser le type `Int`, mais vous **n'avez pas le droit d'utiliser** le type `Integer`[1]. Puisque la capacité du type `Int` est bornée, il faudra utiliser des **listes** de `Int` pour représenter nos entiers. Vous pouvez commencer votre programme avec :

```
type MyInt = [Int]
```

ce qui signifie qu'on déclare un nouveau type qui est un synonyme du type "liste de `Int`". Le type `MyInt` correspond à notre implémentation des entiers sans borne (qui remplace le type `Integer`)

Vous devez programmer **au moins** les fonctions suivantes (vous pouvez bien sûr ajouter d'autres si vous voulez)

1. `printMyInt :: MyInt -> String` fonction qui, étant donné un élément de votre type, retourne une chaîne de caractères qui correspond à l'entier. Ici vous aurez besoin de la fonction `show` de Haskell qui peut transformer un `Int` en `String`.
2. `int2myint :: Int -> MyInt` fonction qui transforme un `Int` en `MyInt`.
3. `addMyInt :: MyInt -> MyInt -> MyInt` : addition
4. `subMyInt :: MyInt -> MyInt -> MyInt` : soustraction
5. `ltMyInt :: MyInt -> MyInt -> Bool` : comparaison (retourne `True` si le premier argument est strictement inférieur)
6. `multMyInt :: MyInt -> MyInt -> MyInt` : multiplication
7. `divMyInt :: MyInt -> MyInt -> MyInt` : division
8. `powMyInt :: MyInt -> Int -> MyInt` : x à la puissance y, où le type de l'exposant y est `Int`.

NB: Pour simplifier votre tâche, vous pouvez supposer qu'on ne va pas représenter des entiers négatifs. Notamment, la fonction `subMyInt` peut retourner une erreur ou une valeur de votre choix si le premier argument est strictement inférieur au deuxième. Pour la fonction `divMyInt` on retourne la partie entière de la division.

Exemple d'utilisation :

```
*Main> x1 = int2myint 1234
*Main> x2 = int2myint 2345
*Main> printMyInt (addMyInt x1 x2)
"3579"
*Main> printMyInt (multMyInt x1 x2)
"2893730"
*Main> printMyInt (divMyInt (multMyInt x1 x2) x1)
"2345"
```

NB2: pour cet exercice on ne se soucie pas trop de l'efficacité de l'implémentation en termes d'utilisation de mémoire. Donc, vous le trouverez peut-être plus facile d'utiliser une implémentation "naturelle" où un `MyInt` est une liste qui correspond à la représentation décimale d'un entier (c'est-à-dire, vous pouvez utiliser des listes dont chaque élément est entre 0 et 9). Ça va gaspiller un peu de mémoire, mais ça vous permettra de lire plus facilement les valeurs de vos variables.

Exemple : l'entier 1234 du programme ci-dessus, peut correspondre à la liste $[1, 2, 3, 4]$ ou bien la liste $[4, 3, 2, 1]$ (vous pouvez choisir de mettre le chiffre le plus important au début ou à la fin).

[1] Le but de cet exercice et de réimplémenter la fonctionnalité de ce type à partir de zéro, donc si vous l'utilisez dans votre programme ça n'a pas de sens !

Vérification

Pour tester que votre programme fonctionne correctement vous devez obligatoirement implémenter les fonctions suivantes :

1. `fact :: Int -> MyInt` où `fact n`=$n!$ (factorielle)
2. `choose :: Int -> Int -> MyInt` où vous utiliserez la formule

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

pour calculer le coefficient binomial des paramètres donnés (donc `choose 7 2` devra retourner $\frac{7!}{2!5!} = 21$).

Exemple :

```
*Main> printMyInt (fact 50)
"30414093201713378043612608166064768844377641568960512000000000000"
*Main> printMyInt (choose 40 20)
"137846528820"
*Main> printMyInt (divMyInt (fact 80) (fact 40))
"87716774664850851422635372244582690394347681205207802985840640000000000"
*Main> printMyInt (foldl (addMyInt ) (int2myint 0) (map (choose 30) [0..30]))
"1073741824"
```

Que fait la dernière ligne ci-dessus ? Hint : rappelez vous de la loi binomiale :

$$\sum_{i=0}^{n} \binom{n}{i} = 2^n$$

Donc, que retourne l'expression suivante ?

```
subMyInt
      (foldl (addMyInt ) (int2myint 0) (map (choose 30) [0..30]))
      (powMyInt [2] 30)
```

Quelques Conseils

1. Pour toutes les opérations vous êtes encouragés à utiliser les algorithmes classiques que vous utilisez pour faire ces opérations à la main. Notamment, pour la multiplication, vous serez peut-être tentés d'implémenter `x*y` comme `y+y+...+y` (avec `x-1` additions). **Cela ne va pas marcher suffisamment bien pour faire les vérifications ci-dessus, car cet algorithme n'est pas du tout efficace !** Vous devez donc implémenter un vrai algorithme de multiplication pour les mêmes raisons que si on vous demande de calculer 123×456 à la main, vous ne ferez pas $123 + 123 + \ldots + 123$, 456 fois. Idem pour la division.
2. Les exemples ci-dessus sont censés vous aider tester que votre programme fonctionne correctement. Votre programme devra retourner les réponses indiquées (dans quelques secondes maximum). Cependant, c'est une bonne idée de faire plus de tests avec des valeurs qui dépassent 2^{63}. Pour le faire, vous pouvez simplement vérifier si ce que votre programme retourne est d'accord avec l'implémentation du type `Integer` de Haskell. Par exemple, pour vérifier qu'on a bien calculé $80!/40!$ (comme donné ci-dessus) on peut écrire :

```
*Main> product [41..80]
87716774664850851422635372244582690394347681205207802985840640000000000
```